# Chapter B



# Errata

---

There is no better way to start a new year, but by starting an Errata for my new book. Of course I am bitter-kidding. The reason I have to start this errata is not a technical one. I am sure that as more people starting reading this book, this errata will get bigger and technical corrections will be added as well. The reason why I needed to start an errata so soon after the book was released is because one of the most dedicated collaborator's name was left out of the **Acknowledgements** section.

I have no explanation and excuse for how this happened, so I'll add his name here. A special thank you for providing a lot of technical corrections for the previous editions of this book goes to **Süleyman Onur**.

## B.1 Book corrections

### Chapter 2: Spring Bean Lifecycle And Configuration

In the **Spring Configuration Classes and the Application Context** section, page 59 the following paragraph is outdated:

---

**X** The `@PropertySource` annotation adds a bean of type `PropertySource` to Spring's environment that will be used to read property values from a property file set as argument. The configuration also requires a bean of type `PropertySourcesPlaceholderConfigurer` to replace the placeholders set as arguments for the `@Value` annotated properties.

---

Configuring properties to be injected with `@PropertySource` no longer requires a bean of type `PropertySourcesPlaceholderConfigurer`. The change was introduced in September 2018, but I missed it. I've been declaring that static bean to support property injection since the first time I did not use it and had to dig into the Spring reference to find out why my configuration doesn't work.
The current configuration in the book is not technically incorrect, just technically redundant. So, I would like to correct the previous affirmation in this errata to:

---

** The `@PropertySource` annotation adds a bean of type `PropertySource` to Spring's environment that will be used to read property values from a property file set as argument. If placeholder syntax needs customization a static bean of type `PropertySourcesPlaceholderConfigurer` can be declared and used for this purpose.[1]

---

*(Observation submitted by Farid Guliyev)*

In the **Constructor Injection** section, page 74 the following paragraph needs some clarification:

---

The `@Autowired` annotation provides an attribute named `required` which, when it is set to `false` declares the annotated dependency as not being required. When `@Autowired` is used on a constructor, this argument can never be used with a value of `false`, because it breaks the configuration. It's logical if you think about it, you cannot make optional the only means you have to create a bean right? We'll come back to this topic when we talk about setter injection.

[1] Javadoc reference: `https://github.com/spring-projects/spring-framework/blob/master/spring-context/src/main/java/org/springframework/context/annotation/PropertySource.java`

The `@Autowired` annotation can be placed **on a constructor** like this:

```
@Component
public class Car {
    private SteeringWheel steeringWheel;

    @Autowired(required = false)
    public Car(SteeringWheel steeringWheel) {
        this.steeringWheel = steeringWheel;
    }
}
```

In this case the `required = false` is ignored, because constructor-injected components are always returned to the client (calling) code in a fully initialized state, this means that injecting the declared dependency is mandatory.[2]
If we try to start a Spring application with that bean configuration and no bean of type `SteeringWheel` is found the application won't start and a `UnsatisfiedDependencyException` will be thrown.
But, the `@Autowired` annotation can also be placed **on a constructor parameter** like this:

```
@Component
public class AnotherCar {
    private SteeringWheel steeringWheel;

    public AnotherCar( @Autowired(required = false) SteeringWheel steeringWheel) {
        this.steeringWheel = steeringWheel;
    }
}
```

In this case a `null` value will be injected, because the constructor parameter is not subjected to constructor injection rules.
Spring can be tricked into accepting a missing dependency for a constructor by using the `Optional<T>` Java type. The following piece of code depicts this exact scenario.

```
@Component
public class OptionalPartsCar {
    private SteeringWheel steeringWheel;

    @Autowired
    public OptionalPartsCar(Optional<SteeringWheel> steeringWheelOpt) {
        steeringWheelOpt.ifPresentOrElse(sw
             -> this.steeringWheel = sw, () -> steeringWheel = null);
    }
}
```

Spring is smart enough to insert a `Optional.empty()` value, but this doesn't mean the the resulting bean is fully-configured and thus, we just bent the rules a little. Also using a parameter of type `Optional<T>` is not really recommended for a lot reasons you can Google yourself.
The code in mentioned here is now a part of the official repo, you can find it in sub-project `chapter02/beans`, the `com.apress.cems.beans.required` package. *(Observation submitted by Farid Guliyev)*

In the **Bean Scopes** section, page 94 the following paragraph is incorrect.

[2]Official documentation reference about the constructor injection: `https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-constructor-injection`

**X** Fun fact: Using `@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)` actually does nothing. Because Spring IoC container is not being told what kind of proxy to wrap the bean in, instead of throwing an error, just creates a singleton. So if you really want to customize scope `proxyMode` attribute must be set.

It should be:

** Fun fact: Using `@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)` does not seem to do very much. Because Spring IoC container is not being explicitly told what kind of proxy to wrap the bean in, it just doesn't. The `proxyMode` attribute has a default value which is `ScopedProxyMode.DEFAULT`, which is equal to `ScopedProxyMode.NO`, unless explicitly configured otherwise. So if you really want to customize scope `proxyMode` attribute must be explicitly set. Otherwise the scope will be applied to the bean, and a `prototype` scoped bean when injected into a `singleton` scoped bean will act just like a singleton bean. Only when a `prototype` scoped bean is being repeatedly requested from the context its scope will become obvious.

And because of the previous affirmation the following paragraph is incorrect as well.

**X** Which value should be used for the `proxyMode` in the previous implementation, to make sure that will always get a fresh new instance when the bean is accessed? If you really want to delegate that decision to Spring, you can use `proxyMode = ScopedProxyMode.DEFAULT` and the container will play it safe and create a CGLIB-based class proxy by default. But if you want to make the decision, just look at the class code. If the class implements an interface, then `proxyMode = ScopedProxyMode.INTERFACES` can be used. But if the class does not implement an interface, the only possible option is `proxyMode = ScopedProxyMode.TARGET_CLASS`.

And should be corrected to:

** Which value should be used for the `proxyMode` in the previous implementation, to make sure that will always get a fresh new instance when the bean is accessed? If you want to make the decision, just look at the class code. If the class implements an interface, then `proxyMode = ScopedProxyMode.INTERFACES` can be used. But if the class does not implement an interface, the only possible option is `proxyMode = ScopedProxyMode.TARGET_CLASS`.

*(Observation submitted by Farid Guliyev)*

In the **Bean Scopes** section, page 100 the following paragraph is slightly incorrect.

**X** Starting with Java 8 interfaces can be declared to contain *private* and *default* methods.

** Starting with Java 8 interfaces can be declared to contain *default* methods and in Java 9 *private* methods were introduced.

*(Observation submitted by Farid Guliyev)*

## Chapter 3: Testing Spring Applications

In the section **Unit Testing Using JUnit** , page 215 the package name is wrong. This section cover JUnit 4 so the package should be `org.junit.Assert` instead of `org.junit.jupiter.api.Assertions`.

In the **Testing Spring Applications** section, page 247 the `SpringUnitTest` class uses a combined approach using JUnit4 and JUnit5 components to create a Spring configuration containing Mockito mock beans. When I was working on the book at the beginning of the year 2019 that test used to work. After upgrading JUnit 5 and Spring to the most recent versions, that test no longer works as expected. I cannot change the code in the book, but I already updated the class on the official GitHub repository to the following implementation:

```
package com.apress.cems.jupiter.mock;
...
@ExtendWith(MockitoExtension.class)
public class SpringUnitTest {
    public static final Long PERSON_ID = 1L;
    PersonRepo personRepo;

    // mocking the database
    @Mock
    JdbcTemplate jdbcTemplate;

    @BeforeEach
    void init() {
        personRepo = new JdbcPersonRepo(jdbcTemplate);
        Mockito.when(jdbcTemplate.queryForObject(anyString(),
             any(PersonRowMapper.class), any(Long.class))).thenReturn(new Person());
    }

    @Test
    public void testFindByIdPositive() {
        assertTrue(personRepo.findById(PERSON_ID).isPresent());
    }
}
```

*(Observation submitted by Farid Guliyev)*

In the **Quick Quiz** section, page 270 the **Question 2** statement could use a revamping to allow for a multiple answer, since the possible answers are mutually exclusive. So in future editions, the statement will be changed to:

---

** Given the following unit test, choose the actions that could be taken from the list of answers to allow the test to be executed correctly. (Choose all that apply.)

---

*(Observation submitted by Farid Guliyev)*

## Chapter 4: Aspect-Oriented Programming with Spring

In the **Defining Pointcuts** section, page 296 there is a statement about the `within(..)` designator that is incorrect. (Don't ask me how I reached to that conclusion!)

---

X There is also a list of designators that can be used to define the reach of the pointcut, for example the `within(...)` designator can be used to limit the pointcut to a package, but it works only with AspectJ.

---

The affirmation should be corrected to:

** There is also a list of designators that can be used to define the reach of the pointcut, for example the `within(...)` designator can be used to limit the pointcut to a package.

*(Observation submitted by Farid Guliyev)*

In the **Defining Pointcuts** section, page 296 there is a statement in the observations box that is incorrect.

X The `[Modifers]` is not mandatory and if not specified defaults to `public`.

The affirmation should be changed to:

** In execution pointcuts, if the `[Modifers]` is not explicitly specified and a `*` is used, otherwise the pointcut expression is incorrect) it applies to all the methods intercepted by the proxy. When interface based proxies are used, it defaults to `public` because all methods in an interface are public by default and implementing classes are not allowed to change the modifier to a more restrictive one. When class based proxies are used (the proxy extends the target class), methods having `public`, `protected` and having no modifier (also known as `default`) are intercepted.

*(Observation submitted by Farid Guliyev)*

In the **After Throwing** section, page 309 there is a statement about this type of advice that needs a more detailed explanation.

** This type of advice does not stop the propagation of the exception, but it can change the type of exception being thrown.

The name of the advice is *after throwing*, so the exception was thrown, that cannot be taken back. This is related to how the JVM handles exceptions. The throwable is thrown from the current method to the JVM, which checks this method for a suitable handler. If not found, the JVM unwinds the method-call stack, looking for the closest calling method that can handle the exception described by the throwable. If it finds this method, it passes the throwable to the method's handler, whose code is executed to handle the exception.

But this exception is provided as an argument to the advice method annotated with `@AfterThrowing`. This method can throw it's own exception which will replace the one currently on the method stack. This behaviour can be considered stopping the propagation of the initial exception, but overall the execution of the intercepted method still results in an exception being thrown. If the advice method does not throw its own exception, the original exception is propagated. Not even a `return` statement in the advice method can stop the exception from propagating. The only thing that can do so is a `System.exit(0);` statement, but that is not a correct behaviour of a proxy.

In the **Around** section, page 309 there is a statement about this type of advice that needs a more detailed explanation.

** The around advice is the most powerful type of advice because it encapsulates the target method and has control over its execution, meaning that the advice decides whether the target method is called, and if so, when and if the result is to be returned and it is the only one with the power to do all this.

This statement could be enriched to explain better why this advice is powerful, because the `@Before` advice

can prevent the execution of the target method by throwing an exception, or can delay the execution by calling `Thread.sleep(..);`. It might look like the `@Around` advice is nothing special. So allow me to make a list with all the things the `@Around` advice can do:

- it can prevent the target method from being called
- it can delay the target method from being called
- it can intercept the result and modify it
- it can intercept the exception thrown by the target method and it can swallow it( stopping it from propagating) making it look like the method execution returns normally
- it can intercept the exception thrown by the target method and it can enrich it with information before propagating it or it can replace it with its own exception

In the **Conclusions** section, page 315 there is a statement that is incorrect.

---

**X** Only **public** Join Points can be advised (you probably suspected that).

---

The affirmation should be corrected to:

---

** Only **non-private** Join Points can be advised (you probably suspected that)..

---

*(Observation submitted by Farid Guliyev)*

## Chapter 5: Data Access

In the **Testing Transactional Methods** section, page 403 there is a statement about the `@Commit` annotation that is not true.

---

**X** The `@Commit` annotation was introduced in Spring 4.2 to modify the default behaviour when needed, and has the same effect as `@Rollback(true)`.

---

The affirmation should be changed to:

---

** The `@Commit` annotation was introduced in Spring 4.2 to modify the default behaviour when needed, and has the same effect as `@Rollback(false)`.

---

*(Observation submitted by Farid Guliyev)*

There are two more references to `@Commit` being equivalent to `@Rollback(true)`, in code snippets on page 404 and 405. No idea where those come from or why they are there, since they are not present in the official sources.